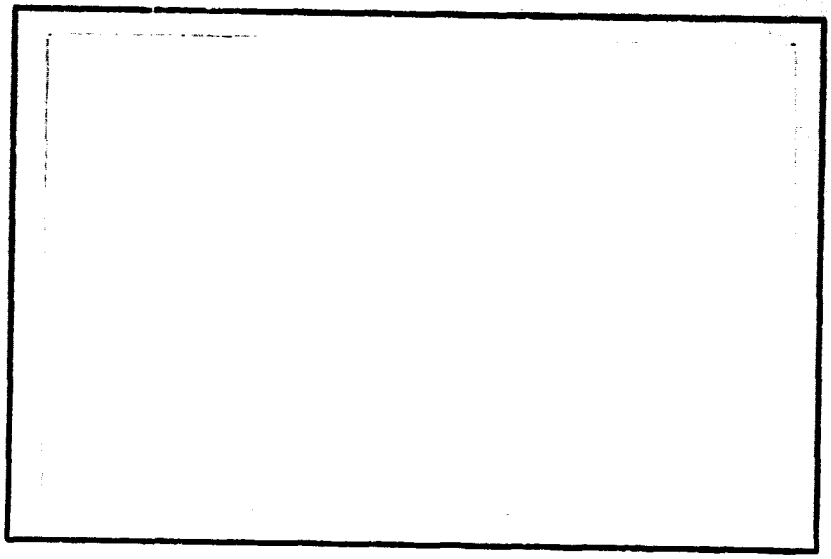
N O T I C E

THIS DOCUMENT HAS BEEN REPRODUCED FROM MICROFICHE. ALTHOUGH IT IS RECOGNIZED THAT CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED IN THE INTEREST OF MAKING AVAILABLE AS MUCH INFORMATION AS POSSIBLE

Appendix B

UNIVERSITY OF COLORADO



(NASA-CR-163420) TOOLS TO AID THE SPECIFICATION AND DESIGN OF FLIGHT SOFTWARE, APPENDIX B (Colorado Univ. at Boulder.) 11 p HC A02/MF A01 CSCL 09B

NSG-30063

Unclas
G3/61 28335

DEPARTMENT OF COMPUTER SCIENCE

Technical Report

TOOLS TO AID THE SPECIFICATION AND DESIGN OF FLIGHT SOFTWARE

by

Guy Bristow
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-168-80 January, 1980

This work was supported by grant NSG 1638
from NASA Langley Research Center

INTRODUCTION

The development of flight software is recognized as an expensive undertaking. Flight software will often have the following characteristics:

1) It is real-time with potentially strict performance constraints upon response times, etc.
2) There can be a large number of independent and very different devices.
3) Likewise, there can be a large number of independent, concurrent activities.
4) The software may run on more than one processor, using more than one language.
5) There must be a very high degree of reliability and recoverability.

These characteristics, when combined, make the software development task formidable and expensive. However, flight software often has a very short lifespan, making the cost of its development disproportionately high. This has prompted much research into ways to reduce this cost given the current state-of-the-art in computer science.

One important way to reduce this cost is to provide a complete software development environment, containing, among other things, automated tools to perform, or aid in the performance of, every task during software development. Such tools would have a common, user-friendly interface, and would operate so as to augment each other in any situations where overlap of tasks occurs. This paper considers the problem of which such tools should be provided.

STAGES OF SOFTWARE DEVELOPMENT

We regard the development of software as roughly falling into three main stages, each of which we further subdivide into two substages.

Definition

Requirements Definition. This is the very early stages of a project in which the tasks that are to be performed by the system are determined.

Specification. In this substage, the requirements are expanded and rigorously defined, possibly introducing hardware considerations, resulting in a document against which the final product can be tested for correctness.

Design

Architecture Design. The high level organization of the final product is delineated during this substage, involving the subdivision of tasks among processes, the interaction between and among those processes, the devices, and other software in the system, and the organization of data shared by processes.

Algorithm Design. During this substage, the structure, data organization, and data-flow paths of individual processes are determined, and the subdivision of those processes into small, rigorously defined, programmable units is accomplished.

Implementation

Program Implementation. This substage encompasses the coding and testing of individual programmable units.

System Implementation. This substage involves the integration of programmable units to form successively larger units, to be tested against successively higher level descriptions of the system, until the completed final system is obtained and is tested against the specification.

This is not a complete list of all the activities that go on during the development of a software system. There are many other activities which occur in parallel with those we have delineated - documentation, communication, project management, etc. There is also the maintenance stage, occurring after all the above stages, but which we regard as the repetition of some or all of the above activities on (presumably) small sections of the system.

Our interest lies primarily in the first two of the stages, namely specification and design. In the Appendix, we have given tables of tools, usable during each of the stages defined here, but the remaining

discussion focuses upon the first two stages.

AGENTS

In addition to identifying tasks and activities that are performed during specification and design, we also identify the agents that perform such tasks and have organized the tables in the Appendix according to the usefulness of various tools to these various agents. The agents do not necessarily have a direct correspondence to people - one person may play the role of several agents, while one agent may in fact consist of a team of people.

SOME GENERAL OBSERVATIONS

A software development environment, as we envisage it, consists of two main components. Firstly, there are languages to describe the system during each of the various stages - specification languages, design languages, etc. Secondly, there are the software tools to support the work performed during each stage, as well as the work that is done to progress from one stage to the next.

This report deals primarily with the second of these components; namely software tools. Inevitably, however, the two components are interrelated, and some language suggestions may appear as well.

Before entering a discussion of which tools should be present, we first give some general considerations on the nature of the tools.

Methodologies. It is now well accepted that the smooth and successful development of a large and complicated software system requires the rigorous imposition of certain methodological principles, practices and procedures. Some of these are so obvious that they go without saying - modular programming, rigorous testing, full documentation, etc. Some are maybe not so obvious, and their effectiveness may well depend on the particular application. We would like, wherever possible, to leave the choice of the methodological aspects of the development environment up to the project managers and leaders. However, once such methodological aspects are chosen, we would like the system to enforce them rigorously. Some examples of such methodological

aspects are: top-down design method, data abstraction, controlled access and update of the system description during development, etc.

Consistency. During each stage of software development, the description can be thought of as going through several levels of refinement. For instance, the first specification that is produced will be necessarily at a high level, and subsequently this is refined and corrected until the final, detailed specification is produced. There are two forms of verification that need to be performed at each level. The first of these is to verify that the description at this level is consistent and unambiguous within itself. The second is to verify that the description correctly and completely matches the description at all levels above, i.e., is consistent with previous descriptions. Throughout this report, these two types of consistency determination are referred to as internal and external verification. For the first (highest) level of description at a particular stage, the external verification compares this description against the descriptions from the previous stages.

Incompleteness. At each level, much work is performed with an incomplete description of the system at that level. For instance, in the early stages of architecture design the designer may wish to experiment with several different overall designs before making the final choice. Requiring complete descriptions for each such design would involve a large amount of essentially unneccessary work. In this case, it is desirable to provide tools that can work with partial descriptions of the architectural design. At each stage, therefore, we would like to see tools that operate on incomplete system descriptions where possible.

Cross-referencing and Tracing. Two important facilities must be provided throughout the entire development process to aid in modifying and maintaining the system. The first of these is a powerful cross-referencing facility that allows for the rapid determination of all the effects of a proposed modification at a particular level on the rest of the system at that level. The second is a powerful tracing tool, that provides links between the different components of the system at different levels. When there is a proposed modification at a previous

(higher) level than the current level, such a tool determines which parts of the lower-level descriptions are affected by such a modification.

Automatic Documentation. One area where much time can be saved by the use of tools is in documentation. By providing tools to generate, summarize and paraphrase the description at each level, not only can the documentor's time be saved, but also the time of readers who may otherwise have to wade through documents containing more information than is needed.

Testing. While much internal and external verification can be done "statically" by providing analysis tools, it is currently beyond the state-of-the-art to have a complete set of static verification tools. Accordingly, testing is still an important approach to error detection and there are various ways that tools can be provided to aid in the testing of software.

Firstly, we would like to see tools that "test" the specification and design descriptions of the systems. Such testing requires simulation systems that animate the description, and symbolic execution systems to detect the errors. Such testing tools would augment automatic error detection and also hand-checking.

Secondly, we would like to see tools that automatically generate test cases from the specification and design descriptions. Such test cases would be used to test parts of the system during module integration/system implementation. Such test cases as are generated from the specification would be used to test the final system. Those generated from the architectural design would be used to test the processes of the system, and so on.

Finally, we would like to see a complete testing environment, providing automatic testing, simulators for the parts of the system not under test, automatic analysis of the test results, and also the storing of test cases and their results. (Such test cases and their results can be later used to check whether modifications to the final system have affected any of its operations).

ACTIVITIES AND TOOLS

Requirements Definition

The definition of requirements may well be written, and will almost certainly be read, by people with limited computer or mathematical backgrounds. To be understood, therefore, it will have to be written in English, presumably loosely structured (see [1]). This rules out the use of analysis tools. Such tools as will be used at this stage will largely consist of editing and report generating tools. Some tools could be provided as aids to hand analysis, e.g., tools to aid in cost estimation, etc.

Specification

The specification is more rigorous than the requirements definition, and will presumably be written in a specification language. This allows for the use of automatic analysis tools. Some specification languages have already been produced, see for example [2] and [3]. Inevitably, some of the specification - relating to the processing performed by the system - will be written as a prose description since no specification language can have enough special operations to describe all possible ways of processing input. Such prose descriptions can be made more rigorous, and potentially analyzable, by the use of such tools as the automatic specification formalizer and action sequencer [1]. The initial creation and updating of the specification will be performed using various text editors, display tools, etc.

Currently, few tools exist that perform analysis on system specifications. This is a real problem, because analysis of program errors reveals that typically a large number are introduced as a result of errors in, or incompleteness of, the specification. We believe that whatever analysis can be performed at each stage should be performed, since the longer that an error remains undetected, the more costly it is to correct.

Since specification tools are not currently available, we instead identify tasks that need to be performed, in the hope that tools

can be developed to perform such tasks. In the Appendix we have shown the relationships of these various tasks, indicated the agents concerned with the task, and indicated which extent tool are of possible utility.

One task is to test for completeness. A complete specification defines exactly a range of legal inputs, such that all inputs outside this range are erroneous. For each legal input, an action is defined. Usually resulting in an output and/or a change of internal state. Where this action depends on the internal state (as evidenced by previous inputs), all such states are accounted for.

Another task is to test for internal consistency. Each input and internal state will usually result in only a single legal action. Should ambiguity exist, this should be detected and reported.

An important task is to test for correctness (external verification). Since the requirements definition will not be in machine understandable form, such testing cannot be by automatic analysis. Instead, we anticipate it will be done with the aid of simulation and modelling. This will provide information about behavior during unanticipated input sequences, about reliability, useability, and the relationship of the system with its environment.

Following the acceptance of the specification, information needs to be extracted from it for various purposes. Some of these purposes are as follows: feasibility studies, cost estimation, standards checking, test data generation, etc. We can further identify various agents who need part or all of the information - e.g., user guide writers, trainers, system designers, project managers etc.

We also anticipate that there will be modifications that must be made to the specification. Here, as in all phases of system development, we suggest the use of a regulatory system to control how and by whom such updates are made. The data base underlying this regulatory system we feel, should, retain previous versions of the specification to allow for the assessment of the impact of modifications.

Initial Architectural Design

Producing the initial architectural design requires the balancing

of several different factors. The way that the factors are balanced will depend on the particular requirements and constraints imposed on the system. Some factors that normally must be considered are as follows:

1) *Accuracy.* This is the degree to which the final system obeys the specifications.
2) *Reliability.* This includes such concerns as the mean time between failures, security features, recoverability, etc.
3) *Cost of development.* Many factors have been found to affect the development cost. Ways to reduce this cost would include:
   a) reducing the total amount of software (number of instructions),
   b) keeping the programmable units simple to code and test,
   c) keeping the overall system simple to test,
   d) reducing the amount of shared data,
   e) reducing the amount of information transferred among system modules, and
   f) producing programmable units that are independent, with simple and uniform interfaces.
4) *Maintainability.* The ease with which the system can be maintained and modified can be increased in several ways, including:
   a) producing functionally independent processes and programmable units,
   b) using the concept of data abstraction,
   c) ensuring data independence,
   d) having the system model the real-world, and
   e) having uniform device access, and inter-process communication methods.
5) *Speed of production.*
6) *Resource usage.* This includes concerns such as usage of main memory, disk space, etc.

We propose the development of a system to aid the architectural designer in deciding between various different designs that he may be considering, and also give him some early estimates of the factors above. This system would allow the designer to specify configurations of processes, the data and accessing paths between them, and the processing that is done by each process. Wherever possible it would produce estimates of the factors above, and where the automatic generation

of such estimates is currently beyond the state-of-the-art, it would allow the designer to insert his own estimates. Such estimates would include reliability (hardware and software), size, complexity, testability, time to program, resource usage, response times, etc. The factors that it would need to consider would include amount of global data, how it is accessed, amount of data sharing and data transfer, functional independence, etc. The system would allow for rapid and easy reconfigurability, and may also perform automatic reconfiguration where possible.

Architecture Design

We anticipate that such a system as proposed above would be used with brief, and possibly incomplete, descriptions of the architectural design. When a design has been finalized and chosen, it will be expanded and made more detailed. This will require the use of other analysis tools to test for internal and external correctness. We now present a list of those tasks we have identified that need to be performed on the architectural design.

Internal Verification. The first task of the internal verification is to check that interfaces are consistent. Where there is communication between two processes, this should be indicated by some form of "channel" in the architectural design model. Checking for consistency reduces to checking that the data format is identical at either end of the "channel."

It will be necessary to test for the absence of deadlock, contention, illegal execution sequences etc. This can be done by modelling (e.g., with DREAM [4]) or by static analysis (e.g., by the techniques given in Bristow [5] or Reif [6]). Such illegal execution sequences would include anomalies concerning shared or global data, thus allowing a check that the data required by a process has been predefined and is available to that process.

External Verification. Much can be done in the way of external verification of the architectural design model. Firstly, the input and output formats can be checked against the input and output formats in the specification. This check could be extended to insure that all

of the inputs/outputs in the specification are included in the architectural design, and that no other (illegal) inputs are possible in the design.

It should be possible to check that each input correctly leads to its corresponding output, with the correct processing. This would require a certain amount of hand checking. A list could be produced of the possible paths through the system for each input, including the processing at each step. The hand checker would verify that only desirable paths are possible, and that the processing on such paths is correct. This could be done with the aid of facilities such as developed for TOPD [7].

Other Tasks. Assuming a design is produced that is internally and externally correct, other information will need to be obtained from it. It may well have to satisfy some form of 'quality control,' and factors that may need to be considered are cost, maintainablilty, resources required, time required for production, reliability, accuracy, and whether it satisfies all preset standards. Estimates will need to be made of these factors, possibly with machine aid.

In addition, we anticipate the automatic generation of tests and test data from the architectural design. This would be part of the same scheme as mentioned for the specification.

We identify a number of agents who would need to extract information from the architectural design. Firstly, there are the algorithmic designers and developers, who need to get a detailed view of a small section of the system. There are the system testers who require to know the overall functionality of the system and its processes. There are maintainers, who need to know how particular sections of the system interact with the rest of the system. There are project managers, who require information about size, speed, cost, progress, etc. And there are others such as documentors, trainers, operators, etc.

CONCLUSION

We have identified many of the tasks that are normally performed, or that we would like to see performed, during the specification and architecture design stages of software development. Wherever possible

we have identified ways that tools could perform, or aid the performance, of such tasks. Much of the verification and analysis that we have suggested is currently rarely performed during these early stages, but is is our belief that this analysis should be done as early as possible so as to detect errors as early as possible.

REFERENCES

1. R. Balzer, N. Goldman and D. Wile. Informality in program specifications. *IEEE Trans. on Software Engineering, SE-4*, 2 (March 1978), 94-103.
2. D. T. Ross and K. E. Schoman. Structured analysis for requirements definition. *IEEE Trans. on Software Engineerin, SE-3*, (January 1977), 6-15.
3. N. M. Goldman and D. S. Wile. A data base foundation for process specification. Tech. Report, Info. Sci. Inst., Marina del Rey, California (October 1979).
4. W. E. Riddle, J. C. Wileden, J. H. Sayler, A. R. Segal and A. M. Stavely. Behavior modelling during software design. *IEEE Trans. on Software Engineering, SE-4*, 2 (July 1978), 283-292.
5. G. Bristow, C. Drey, B. Edwards and W. Riddle. Anomaly detection in concurrent programs. *Proc. Fourth International Conf. on Software Engineering*, Munich, Germany, (September 1979), pp. 265-273.
6. J. H. Reif. Analysis of communicating processes. TR30, Comp. Sci. Dept., Univ. of Rochester, New York, (May 1978).
7. P. Henderson. Finite state modelling in program development. *SIGPLAN Notices, 10*, 6 (June 1975), 221-227.

APPENDIX

REQUIREMENTS

PHASE	SUBPHASE (tasks)	AGENT	TOOLS
First Draft	construction (insertion, retrieval, modification)	requirements definers	text processors, editors, graphical displayers/inserters, and languages
	retrieval, correction, addition	requirements definers	(same as above)
	extraction (paraphrase)	customers	standards checkers
	assessment (animation)	customers	modelling systems
	assessment (inference)	customers integrators	completeness and feasibility checkers
Acceptance →			
	extraction (paraphrase)	specification designers users userguide writers trainers	
Subsequent Modifications	proposal for a modification	customers users integrators designers, etc.	text processors, editors, graphical displayers/inserters, languages
	retrieval, correction	(same as above)	(same as above)
	extraction (paraphrase)	requirements definers	(same as above)
	modification	requirements definers	
	assessment (animation)	customers	modelling systems
	assessment (inference)	customers integrators	completeness checkers, simulators
Acceptance →			
	insertion	requirements definers	

SPECIFICATIONS

PHASE	SUBPHASE (tasks)	AGENT	TOOLS
First Draft	construction	specification definers	text processors, language macros, preprocessors, editors, and graphical displayers/inserters
	retrieval, correction, addition	(same as above)	(same as above)
	extraction (paraphrase)	specification definers	editors, text processors
		managers customers	feasibility and standards checker, and cost assessors
	assessment (animation)	specification definers	simulators, modelling systems, and testers
		customers integrators	
	extraction (paraphrase)	designers users userguide writers trainers	
Subsequent Modifications	identical to REQUIREMENTS		

ARCHITECTURE DESIGN

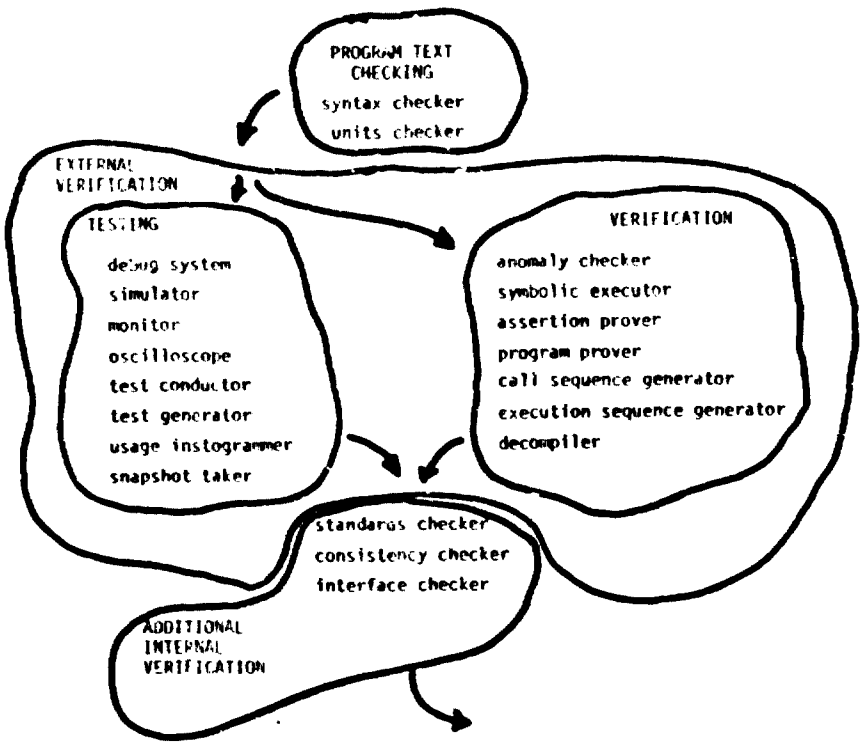
PHASE	SUBPHASE (task)	AGENT	TOOLS
First Draft	construction (insertion, retrieval, modification)	designers	flowchart generators languages, text processors, editors, compilers, and graphical display interaction systems
	retrieval, correction, addition	designers	(same as above)
	extraction (paraphrase)	requirements definers managers	feasibility and standards checkers, cost simulators, modelling systems, symbolic executors, and testers
	assessment (animation)	requirements definers	
	assessment (inference)	requirements definers	translators, consistency checkers (above, within), completeness checkers, and anomaly detectors
Acceptance →			
	extraction (paraphrase)	implementors	translators, transformers, and extractors
		maintainers managers	(same as above)
Subsequent Modifications	proposal for modification	requirements definers integrators implementors	text processors, etc.
	extraction (paraphrase)	designers	(same as above)
	modification	designers	
	assessment (animation)	requirements definers proposal specifier	(same as above)
	assessment (inference)	requirements definers proposal specifier	(same as above)
Acceptance →	insertion	manager	feasibility and cost checkers

ALGORITHM DESIGN

PHASE	SUBPHASE (tasks)	AGENT	TOOLS
First Draft	construction (insertion, retrieval, modification)	designers	languages, text integrators, editors, flowchart generators and displayers, and compilers
	retrieval, correction, addition	designers	debuggers
	extraction (paraphrase)	requirements definers managers	feasibility checkers, cost assessors, and standards checkers
	assessment (animation)	requirements definers	simulators, modelling systems, testers, and symbolic executors
	assessment (inference)	requirements definers	translators, consistency checkers (above, within), completeness checkers, anomaly detecter, symbolic executors, and cross-reference generators
Acceptance	extraction (paraphrase)	implementors maintainers managers	
Subsequent Modifications	same as for ARCHITECTURE DESIGN		

PROGRAM IMPLEMENTATION

PHASE	TOOLS
Module Design	text editors flowchart generators etc.
Coding	text editors language macroexpander structured language preprocessor text formatter translator link editor/loader interpreter subroutine library file system database system operating system
Debugging	syntax checker anomaly checker debug system
Testing and Verification	test conductor simulator test generator monitor decompiler usage histogrammer snapshot taker assertion prover symbolic executor call sequence generator execution sequence generator osilloscope standards checker units checker consistency checker interface checker



Relationships among tools supporting debugging, testing and verification phases occurring during module development.

SYSTEM IMPLEMENTATION

PHASE	SUBPHASE (tasks)	AGENT	TOOLS
Module Integration	integration	developer	debug system (aimed at higher level)
	recursive module redevelopment	developer	as before
System Testing		developers	simulators generators data base analyzer drivers monitors interrupt analyzer linkage editor MAP program program flow ana - lyzer tracer rest result pro- cessor
Verification			
Maintenance		maintainers	(same as above)
Documentation		documentors	editors